



Тестирование программного кода



Общие понятия

Модульные тесты — это сегменты кода, которые проверяют работу других частей кода в приложении, например изолированных функций, классов и т. д. Если приложение успешно проходит все модульные тесты, то вы по меньшей мере уверены, что все низкоуровневые функции работают правильно.



Прочие виды тестирования

- Анализ покрытия кода
- Анализ нагрузки и производительности
- Проверка на стандарты PEP-8
(Pylint, PyChecker, PyFlakes, pep8, coala)

Test Driven Development





Методология TDD

TDD — Test Driven Development. TDD — это методология разработки ПО, которая основывается на повторении коротких циклов разработки:

- изначально пишется тест, покрывающий желаемое изменение
- затем пишется программный код, который реализует желаемое поведение системы и позволит пройти написанный тест.
- затем проводится рефакторинг написанного кода с постоянной проверкой прохождения тестов.



Плюсы тестирования:

- тесты проверяют корректность кода;
- тесты позволяют безопасно изменять код даже в больших проектах.

Минусы тестирования:

- написание тестов требует времени;
- очень часто получается, что в проекте становится больше тестов чем самого кода;
- работающие тесты не гарантируют корректность выполнения кода.

Unit testing tools

Инструмент	Источник	Описание	Автор
unittest	Python standard lib	Первый unit test фреймворк, включенный в стандартную библиотеку.	Steve Purcell
doctest	Python standard lib	Удобны для использования в терминале. Могут интегрироваться с системой epydoc.	Tim Peters
pytest	На основе pylib	Не имеют API, автоматическая сборка тестов, простые asserts, поддержка управления через hooks, кастомизированные трейсбэки.	Holger Krekel
nose	--	Надстройка над unittest. Интерфейс напоминает py.test, но более дружелюбный. Имеет много плагинов расширений.	Jason Pellerin
testify	--	Модульная тестовая платформа с расширениями, батареи сплит – тестов для распараллеливания, поддерживает стандарты PEP8 и специальный менеджер с большим количеством параметров логгирования.	Yelp team
subunit	--	Запуск тестов в отдельных процессах, Отслеживание результатов в единой интегрированной среде	Robert Collins
Sancho	MEMS Exchange tls	Самостоятельно запускает тесты и сохраняет результаты тестов. Используется для систем, которые не должны немедленно реагировать на ошибки.	MEMS and Nanotechnology Exchange

<https://wiki.python.org/moin/PythonTestingToolsTaxonomy>

Начнем путь со стандартного фреймворка
unittest



unittest



Задача

Предположим что мы написали следующий код:

```
# файл cube.py

from typing import Union

def cube_area(side: Union[int, float]) -> Union[int, float]:
    """
        Функция вычисляет площадь поверхности куба
    """
    return 6*side**2

# Стали тестировать и довольны результатом

print(cube_area(3))           → 54
print(cube_area(3.0))        → 54.0
```



Но вот незадача, пришел пользователь...

```
from typing import Union

def cube_area(side: Union[int, float]) -> Union[int, float]:
    """Функция вычисляет площадь поверхности куба"""
    return 6*side**2

side_list = [10, 0, -3 , True , 'five', [1]]
mess = "Площадь поверхности куба для стороны {side} равна: {result}"

for side in side_list:
    result = cube_area(side)
    print(mess.format(side=side, result=result))
```



А: Что делать ?

В: Писать тест !



Пишем тест.

Создем файл с префиксом , постпрефиксом test_имя.py
или имя_test.py

#В нашем случае файл с названием test_cube_area.py

```
import unittest  
from cube import cube_area
```

```
# создаем класс для тестирования нашей функции  
class TestCubeArea(unittest.TestCase):
```

```
    # метод начинаем с префикса test_ . Проверим  
    сначала на идентичность
```

```
    def test_cube_area(self):  
        self.assertEqual(cube_area(3), 54)  
        self.assertEqual(cube_area(0), 0)
```



Запускаем

Флаг `-m` обозначает что нужно запустить файл как модуль

Запуск всех тестов в модуле `test_cube_area.py`

\$ `python -m unittest test_cube_area.py`

Запуск теста из класса `TestCubeArea`

\$ `python -m unittest test_cube_area.TestCubeArea`

Запуск конкретного теста из класса `TestCubeArea`

\$ `python -m unittest test_cube_area.TestCubeArea.test_cube_area`

Запуск всех модулей теста в текущей папки

\$ `python -m unittest`

Проверим на передачу 0 в функцию.

```
import unittest
from cube import cube_area

class TestCubeArea(unittest.TestCase):

    def test_cube_area(self):
        self.assertEqual(cube_area(3), 54)

    def test_value(self):
        self.assertRaises(ValueError, cube_area, 0)

# Запустим тест
$ python -m unittest test_cube_area.py
```



Получим результат:

=====

FAIL: test_value (test_cube_area.TestCubeArea)

Traceback (most recent call last):

File

"/home/vitaliy/www/course/students/test_cube_area.py", line
11, in test_value

self.assertRaises(ValueError, cube_area, 0)

AssertionError: ValueError not raised by cube_area



Встраиваем обработку 0 в код

```
from typing import Union


def cube_area(side: Union[int, float]) -> Union[int, float]:
    """Функция вычисляет площадь поверхности куба"""

    if side == 0:
        raise ValueError("Передано нулевое значение")


    return 6*side**2

# Повторный запуск теста проходит
```

Что в коде нужно поменять ?



Guido van Rossum has strongly opposed adding **exceptions** to the type hinting spec, as he doesn't want to end up in a situation where exceptions need to be checked (handled in calling code) or declared explicitly at each level.



На последок добавим явную проверку типов входящих данных

```
import unittest
from cube import cube_area

class TestCubeArea(unittest.TestCase):
    def test_cube_area(self):
        self.assertEqual(cube_area(3), 54)

    def test_value(self):
        self.assertRaises(ValueError, cube_area, 0)

    def test_types(self):
        self.assertRaises(TypeError, cube_area, True)
        self.assertRaises(TypeError, cube_area, [0])
        self.assertRaises(TypeError, cube_area, {})
        self.assertRaises(TypeError, cube_area, (1, 2))

# Запустим тест и обнаружим что тест не прошел.
# Почему ?
```



Вывод.

Подход к тестированию заставил нас проверить функцию на все допустимые вхождения и усовершенствовать реализацию кода. Если в дальнейшем мы будем менять что-либо то запуская тест мы будем контролировать неизменность работы данной функции.



Пакет pylint

Установим пакет

```
$ pip install pylint
```

```
import sys
class CarClass:
    def __init__(self, color, make, model, year):
        self.color = color
        self.make = make
        self.model = model
        self.year = year
        if "Windows" in platform.platform():
            print("You're using Windows!")
        self.weight = self.getWeight(1, 2, 3)

    def getWeight(this):
        return "2000 lbs"
```



Проверка с помощью pylint

Проверим с помощью pylint

```
$ pylint crummy_code.py
```



Результат

crummy_code.py:15:0: **C**0304: Final newline missing (missing-final-newline)

crummy_code.py:1:0: **C**0114: Missing module docstring (missing-module-docstring)

crummy_code.py:3:0: **C**0115: Missing class docstring (missing-class-docstring)

crummy_code.py:10:24: **E**0602: Undefined variable 'platform' (undefined-variable)

crummy_code.py:12:22: **E**1121: Too many positional arguments for method call (too-many-function-args)

crummy_code.py:14:4: **C**0116: Missing function or method docstring (missing-function-docstring)

crummy_code.py:14:4: **C**0103: Method name "getWeight" doesn't conform to snake_case naming style (invalid-name)

crummy_code.py:14:4: **E**0213: Method should have "self" as first argument (no-self-argument)

crummy_code.py:3:0: **R**0903: Too few public methods (1/2) (too-few-public-methods)

crummy_code.py:1:0: **W**0611: Unused import sys (unused-import)

Условные обозначения

- С – конвенция (convention)
- R – рефакторинг (refactor)
- W – предупреждение (warning)
- E – ошибка (error)


```
"""Модуль для демонстрации """
```

```
import sys
```

```
from datetime import datetime
```

```
from typing import Optional, Union
```

```
class CarClass:
```

```
    """Класс для сущности автомобиль """
```

```
    def __init__(self, color: str, make: Optional[datetime], model: str, year: int,  
type: int) -> None:
```

```
        self.color = color
```

```
        self.make = make
```

```
        self.model = model
```

```
        self.year = year
```

```
        if "Linux" == sys.platform:
```

```
            print("You're using Linux!")
```

```
        self.weight = self.get_weight(type)
```

```
    def get_weight(self, type: int) -> Union[int, None]:
```

```
        """Функция возвращает вес по типу авто """
```

```
        if type == 1:
```

```
            return 2000
```

```
        return None
```

Итог

Линтер помогает делать код отвечающим стандартам языка и соглашениям PEP8.